LLNL-CONF-676764

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

# Puffin: An Embedded Domain-Specific Language for Existing Unstructured Hydrodynamics Codes

C. Earl

August 31, 2015

Fifth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing
Austin, TX, United States
November 16, 2015 through November 16, 2015

**Disclaimer**

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

# Puffin: An Embedded Domain-Specific Language for Existing Unstructured Hydrodynamics Codes

Christopher Earl
Lawrence Livermore National Laboratory
earl2@llnl.gov

## ABSTRACT

In this paper, we present Puffin, a domain-specific language embedded in C++98 for incremental adoption in existing unstructured hydrodynamics codes. Because HPC systems with heterogeneous architectures (traditional CPUs, GPUs, Xeon Phis, etc.) are becoming increasingly common, developers of existing HPC software projects need performance across multiple architectures. While Puffin is not yet complete and only supports CPU execution so far, our aim for Puffin is to provide performance portability to existing unstructured hydrodynamics simulation projects. Our preliminary results focus on two topics. First, we show what the costs of using Puffin are. Adopting Puffin has a initial cost of rewriting existing code into Puffin. Using Puffin has the ongoing costs of increased compilation times (2–3X slower) and runtime overhead (0–11% slower). Second, we show the current benefits of using Puffin and mention the potential future benefits. We show how Puffin can gradually be adopted into an existing project, by doing so with the existing test application, LULESH 2.0. We show a reduction in code length by porting code to Puffin.

## 1. INTRODUCTION

Traditionally, high-performance computing (HPC) projects have code written at a low-level to avoid inefficiencies in high-level code, such as abstraction overhead. While this low level of abstraction ties a project to a specific architecture family, most commonly single-core CPU, for many years HPC hardware has matched these assumptions quite well. In recent years, however, other architecture families, such as multi-core CPUs, many-core CPUs (Xeon Phi) and GPUs, have become increasingly common. With the rise of multiple architecture families, HPC projects are facing a new challenge. Users of HPC projects want to run the projects on multiple platforms, with vastly different architectures, and expect the projects to perform well everywhere.

Fortunately, domain-specific languages (DSL) do not require the high cost that is generally assumed, and can provide the portability and performance necessary for HPC projects.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*WOLFHPC2015* November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4016-8/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2830018.2830021

This paper presents Puffin, an efficient and portable DSL embedded in C++ (the 1998 standard), for unstructured hydrodynamics simulation projects. Puffin is a descriptive language: Code written in Puffin describes what calculation to do without specifying how it should be done. This separation between what and how is very powerful. Domain experts using Puffin can focus on what calculations they want to perform and the correctness of those calculations, without worrying about implementation details. How to efficiently perform calculations can vary drastically between architecture families, which Puffin's abstractions are designed to handle. By writing code that only contains what should be done, how the computation will be carried out can be supplied on an architecture to architecture basis.

Puffin is incrementally adoptable. Hydrodynamics simulation projects that decide to adopt Puffin are not making an all or nothing choice. Once Puffin has been adopted, developers can adopt Puffin at their own pace, and continue new development on the project without using Puffin, for as long as that development makes sense. For example, if a new critical feature is nearing completion just as Puffin is adopted into the project, the current implementation can be finished without Puffin and be ported to Puffin at a later time. Thus, the feature can begin production use without a delay caused by the adoption of Puffin.

Currently, Puffin is not fully implemented. Enough of Puffin has been implemented to produce preliminary results. We use LULESH 2.0 [6] as the first project to port to Puffin. LULESH 2.0 is an update of LULESH [1], a proxy application, which in a short implementation represents the numerical algorithms, data motion, and programming style typical of production-scale software simulation projects. We chose LULESH 2.0 because it is a widely available approximation of production hydrodynamics projects, which is Puffin's core domain.

As Puffin is presently implemented, three fourths of LULESH 2.0's loops can be rewritten in Puffin. The Puffin version of these loops can be written in roughly half the number of lines of code that the original version required. We present several versions of LULESH 2.0 ported to Puffin, with and without recent unoptimized features of Puffin, and with LULESH 2.0's original loop structure as well as with an optimized loop structure. While the Puffin ports of LULESH 2.0 do have up to 11% runtime overhead and compilation 2–3X slower than the original version of LULESH 2.0, they are shorter and potentially more portable than the original version.

The rest of this paper is organized as follows: Section 2 describes the features and capabilities of Puffin that have been implemented. Section 3 describes Puffin's current implementation and optimizations. Section 4 describes how existing hydrodynamics simulation projects can adopt Puffin. Section 5 describes the preliminary results we have with our partial implementation of Puffin and partial port of LULESH 2.0. Section 6 discusses other projects similar to Puffin. Section 7 covers our plan for future development on Puffin. Section 8 contains our conclusions.

## 2. LANGUAGE SEMANTICS

Like most languages, Puffin code can be broken into two categories: Statements and expressions. Unlike many domain-specific languages, Puffin does not define the types of, sizes of, or relationships between basic concepts, such as nodes, elements, and faces. Instead, Puffin provides simple mechanisms for users to define the types and sizes of the basic concepts they need as well as defining the relationships between these concepts.

### 2.1 Aspects and Arrays

In Puffin, arrays of values are the basic unit of storage. Puffin supports arrays of boolean values, integers, floating point numbers, and double-precision floating point numbers. Unlike the common terminology for arrays, in Puffin, we do not refer to an array's *dimensions*, but instead an array's *aspects*. In Puffin's target domain, hydrodynamic simulations, dimensions most commonly refers to physical/spatial dimensions. To avoid confusion with spatial dimensions, we call the dimensions over which Puffin arrays are indexed **aspects**. In particular, spatial dimensions are a single aspect of some Puffin arrays.

Users of Puffin define the aspects they will use for their arrays. To define an aspect, a user needs to provide three pieces of information: A unique nonnegative integer for use as identifier; the size of the aspect, if it has a fixed size; and the storage style for arrays that use this aspect.

In the Puffin port of LULESH 2.0, there are eight aspects. For brevity, below are the definitions for Nodes and (spatial) Dimensions:

```
typedef PuffinAspect<0, −1, Array> NodeAspect;
typedef PuffinAspect<1, 3, Container>
        DimensionAspect;
```

These aspects can be used to define arrays over nodes and dimensions: `NodeDimArray` and `DimNodeArray`. When defining arrays over multiple aspects, the order of the aspects matters. `NodeDimArray` is an array of structures, where the array has a structure for each node and where each structure contains three values, one each for x, y, and z. Conversely, `DimNodeArray` is a structure of arrays, where the structure contains three arrays, on each for x, y, and z, and where each array contains a single value for each node. Since both of these array types are defined over spatial dimensions and number of nodes, objects of either type can be used interchangeably in Puffin.

To use an aspect in a calculation, it must be instantiated:

```
NodeAspect Nodes(domain.numNodes());
DimensionAspect Dim();
```

`Nodes` requires its size as a constructor argument, since it was not defined as fixed-sized. `Dim` does not have any constructor arguments because it was defined as fixed-sized.

Instantiated aspects are used in Puffin statements and expressions to define the range of calculations. This range in turn defines the number of iterations a Puffin statement or expression executes when run. Each iteration of a calculation has a specific index for each aspect in the calculation, which together form an index object. Section 3.1 has more details on index objects. The order of iterations is unspecified to allow for parallelization and future optimizations.

### 2.2 Statements

The previous section describes Puffin's abstractions over data: Arrays and aspects. This section and the following ones describe Puffin's abstractions over calculations and control flow: Statements and expressions.

Puffin statements change state when executed, usually changing values in an array; Puffin expressions produce values and have no side effects. Puffin statements are Puffin's iteration/loop constructs, and each statement explicitly contains the aspects which apply to the statement. When executed, each Puffin statement iterates over all given aspects.

Puffin statements come in two varieties: Standalone and statement blocks. Standalone statements generally only make a single change to state. For example, standalone assignment only assigns values to a single Puffin array. Puffin has three types of standalone statements: Assignments (Section 2.3), reductions (Section 2.6), and assertions (Section 2.7).

#### *2.2.1 Foreach Statements/Blocks*

While Puffin's standalone statements have a lot of expressive power, they are limited to a single change of state per loop. Multiple changes with standalone statements require multiple loops. Puffin statement blocks can avoid the inefficiencies and loop overhead of multiple loops.

A statement block is a series of statements that are performed together in a single loop. Combining multiple statements into a single block/loop shares the loop overhead between all statements. We call individual statements in a statement block dependent statements to distinguish them from standalone statements. All statement blocks begin with Puffin's foreach construct, `puffin_foreach`. The aspects, over which the whole block ranges, follows immediately after the call to `puffin_foreach` in parentheses. After the aspects, the dependent statements follow. With a few exceptions, all dependent statements appear in parentheses. A single call to the member function `execute` with no arguments must end a statement block. Consider this simple example:

```
puffin_foreach(Nodes)
  (a |= 0.0)
  (b |= 1.0)
  .execute();
```

where `|=` is the dependent assignment operator, and `a` and `b` are Puffin arrays over the node aspect. All examples in this section use dependent assignment statements (the `|=` operator) and scalar-value expressions, which are explained in Sections 2.3 and 2.4 respectively.

`puffin_foreach` blocks *without* the `execute` call can be used as dependent statements; consider the following example:

```
puffin_foreach(Nodes)
  (a |= 0.0)
  (puffin_foreach(Dim)
    (c |= 2.0))
  (b |= 1.0)
  .execute();
```

where `a` and `b` are Puffin arrays over the node aspect and `c` is a Puffin array over the node and spatial dimensions aspects. This block performs these assignments in order, so that all of `a`, `b` and `c` are assigned the correct values, despite the differences in aspects.

Additionally, for some dependent statement blocks, particularly when used with conditional statements, there are no extra aspects to add. (For more on conditions, see Section 2.5.) In this case, **`puffin_block`** is used instead of **`puffin_foreach`**; consider:

```
puffin_foreach(Nodes)
  (a |= 0.0)
  (puffin_block
    (c |= 2.0)
    (d |= 3.0))
  (b |= 1.0)
  .execute();
```

where `a`, `b`, `c`, and `d` are Puffin arrays over the node aspect.

## 2.3 Assignment

Puffin assignments are the most common Puffin statement. Consider this simple loop taken from LULESH 2.0:

```
for (Index_t i = 0 ; i < numElem ; ++i){
    sigxx[i] =
    sigyy[i] =
    sigzz[i] = - domain.p(i) - domain.q(i);
 }
```

This loop clearly iterates over both elements and spatial dimensions. By properly defining the element aspect (`Elems`), the spatial dimensions aspect (`Dim`), and the Puffin arrays `sigdd` over elements and spatial dimensions, `p` and `q` over elements, the following Puffin standalone statement can be used instead:

```
sigdd[Elems][Dim] <<= - p - q;
```

For standalone assignment, Puffin uses the operator $<<=$ instead of operator $=$ to make this use of Puffin clearly explicit (and easily searchable). The right-hand side of a Puffin assignment statement is any Puffin expression for which the aspect requirements are satisfied by the aspects on the left-hand side. (Puffin expressions are discussed in Section 2.4.) The left-hand side of a Puffin assignment statement requires some explanation: The left-hand side of a Puffin assignment statement must begin with a Puffin array. Using the array subscript operator (square brackets), aspects to range over are applied to the Puffin array on the left-hand side. For standalone assignment, the aspects used must exactly match the aspects over which the left-hand side array has been defined. In the above example, the whole calculation ranges over both elements and spatial dimensions, which matches the definition of `sigdd`.

As seen in the explanation of `puffin_foreach` above, Puffin uses the operator `|=` for dependent assignment, which is used for general clarity and to distinguish it from the standalone assignment operator. The above example can be written using Puffin's dependent assignment statement as:

```
puffin_foreach(Elems)(Dim)
  (sigdd |= - p - q)
  .execute();
```

Aspects on the left-hand side of Puffin's dependent assignment statement are optional. However, all aspect requirements of both the left- and right-hand side must be satisfied for a dependent assignment statement to successfully compile and execute.

## 2.4 Expressions

The previous section describes Puffin assignment. This section describes Puffin expressions which are used to calculate values. Expressions are used on the right-hand side of an assignment statement, in conditions, reductions, and assertions. See Sections 2.3, 2.5, 2.6, and 2.7 respectively for more details. This section focuses on Puffin's simplest expressions: Arrays, scalars, constants, operators, and pointwise functions. Sections 2.5 and 2.8 describe Puffin's other, more complex expressions.

Like traditional expressions in most programming languages, a Puffin expression calculates and return a value based on its context. In the generic expression, $a + b$, the value returned by evaluating this expression depends upon the meaning of $a$ and $b$, which are generally implied by the context where the expression is used. Puffin expressions rely on context for variable definition/bindings and additionally on index objects. Index objects are discussed in more detail in Section 3.1. Puffin expressions are defined over aspects, which must be present in the index object for the expression to return a meaningful value.

### 2.4.1 *Arrays, Scalars, and Constants*

As implied by the example in Section 2.3, any Puffin array can be used as a Puffin expression. Of course, a Puffin expression that is a Puffin array requires all of the aspects over which it is defined. Additionally, scalar values may be Puffin expressions, which have no aspect requirements. Puffin supports boolean values, integers, floating point numbers, and double-precision floating point numbers as scalar values. These scalar values can variables or constants.

Currently, Puffin supports compile-time inlining of integer values through the use of Puffin constants. A Puffin constant is a cross between a Puffin scalar and a Puffin array: A Puffin constant can be a scalar value or be defined over one or more fixed-sized aspects. All values in a Puffin constant must be fixed at compile-time. Puffin constants serve two main purposes in the Puffin port of LULESH 2.0: First, Puffin constants can serve as bitwise masks to help determine boundary conditions, and second, they store fixed integer coefficients ($\pm 1$) that vary over two aspects.

### *2.4.2 Operators and Point-wise Functions*

Beyond arrays and scalars, Puffin allows for expressions based on mathematical operators and functions. These operators and functions take arbitrary Puffin expressions as operands, so there is no limit on how complex a Puffin expression can be. Puffin expressions based on all of the operators and functions described in this section require the union of their operands' required aspects. Also, since the operators described in this section are overloaded C++ operators, standard C++ operator precedence is used.

Puffin supports the following mathematical operations: addition, subtraction, multiplication, division, negation, square, square root, cubic root, maximum of two values (max), minimum of two values (min), and absolute value. These operations promote value types as needed, following standard C++ type promotion rules (boolean to integer, integer to float, and float to double). This list based off of current use in the Puffin port of LULESH 2.0, and can be easily extended to support other functions such as cube and sine.

Additionally, Puffin supports comparison operators for use in conditions: equal, not equal, less than, less than/equal, greater than, and greater than/equal. Likewise, Puffin supports logical operators: and and or. Unlike the mathematical operations above these comparison and logical operators return boolean values regardless of their operands. Puffin also supports bitwise operators for integer values only: bitwise and and bitwise or. For more information on conditions, see Section 2.5.

Finally, Puffin also support user-defined scalar functions without side effects. We leave out the details of these functions due to space constraints, but briefly users can define functions that take in an arbitrary number of scalar values and produce a scalar value. Currently, users must also define a class which contains their scalar function as well as some boilerplate functions and type definitions.

## 2.5 Conditional Expressions and Statements

In C++, conditional expressions and statements cannot be overloaded: The ternary operator, `if`, and `switch`. Thus, in Puffin we must define our own conditional expressions and statements. For traditional branching, Puffin provides the dependent statements `when` and `whenelse`, explained below in Section 2.5.1. For conditional expressions, Puffin provides `cond`, explained in Section 2.5.2. For optimization and similarity to existing code, Puffin provides the expression `match`, which is roughly analogous to C's `switch` statements and is explained in Section 2.5.3.

### *2.5.1 Conditional Statements*

The dependent statements `when` and `whenelse` provide users with a way to branch differently on each iteration. The `when` statement takes two arguments, a condition and a result. Consider this example from the Puffin port of LULESH 2.0:

```
puffin_foreach(CurReg)
  .when(vnewc <= eosvmin,
        compHalfStep |= compression)
  .execute();
```

where `CurReg` is the region aspect (a subset of all elements), `eosvmin` is a scalar value, and `vnewc`, `compHalfStep`, and `compression` are Puffin arrays over the element aspect. The dependent statement is only executed on iterations for which the condition is true.

The `whenelse` statement takes three arguments, a condition, a true result, and a false result. Consider this example from the Puffin port of LULESH 2.0:

```
puffin_foreach(CurReg)
  .whenelse(vnewc >= eosvmax,
            puffin_block(p_old |= 0.0)
                        (compression |= 0.0)
                        (compHalfStep |= 0.0),
            p_old |= p)
  .execute();
```

where `CurReg` is the region aspect (a subset of all elements), `eosvmax` is a scalar value, and `vnewc`, `p_old`, `compression`, and `p` are Puffin arrays over the element aspect. The first dependent statement is only executed on iterations for which the condition is true, and the second dependent statement is only executed on iterations for which the condition is false.

Finally, we should note that the `when` and `whenelse` dependent statements use member function syntax rather than function call syntax.

### *2.5.2 Arbitrary Conditional Expressions*

While the `when` and `whenelse` dependent statements provide a way to build arbitrarily-complex conditional statement blocks, Puffin also provides a simpler way to write conditions for producing values for a single assignment or other statement. In the `whenelse` example of the last section, the `whenelse` allowed for multiple assignments that differed between the two cases. However, consider the following example where the assignment of `p_old` was the only assignment:

```
puffin_foreach(CurReg)
  .whenelse(vnewc >= eosvmax,
            p_old |= 0.0,
            p_old |= p)
  .execute();
```

This is a fairly complex statement block for a single assignment. The Puffin conditional expression, `cond`, makes this code much less complex:

```
p_old[CurReg] <<= cond(vnewc >= eosvmax, 0.0)
                      (p);
```

There are many variations on `cond` in different functional languages, and Puffin's `cond` uses a simple clause style and semantics compared to most of these. Like all versions of `cond`, Puffin's version allows for an arbitrary number of clauses/conditions. Each non-final clause must contain exactly two arguments, and the final clause must contain exactly one argument. The first argument of a non-final clause is a condition and must be a Puffin expression that returns a boolean value at each iteration. The second argument of non-final clause and the only argument of a final clause must be a Puffin expression. During the evaluation of a `cond` expression in each iteration, the conditions of non-final clauses are evaluated in order. When a condition evaluates to false, the next condition is tried. When a condition evaluates to true, the expression associated with that condition is evaluated, and its result is returned as the result for the `cond` expression for that iteration. If all conditions evaluate to false, then the expression in the final clause is evaluated, and its result is returned as the result for the `cond` expression for that iteration. The general form is:

```
cond(condition_1, expression_1)
    (condition_2, expression_2)
    ...
    (final_expression)
```

### 2.5.3 Integer-Comparison Conditional Expressions

Puffin provides one other conditional expression, which is far more restrictive than `when`, `whenelse`, or `cond`: `match`. `match` is roughly analogous to C's `switch` statements, with the main exception that `switch` is a statement allowing arbitrary side-effects, and `match` is an expression and only returns values.

The syntax for `match` is:

```
match(integer_expression)
  .pcase<case1>(expression1)
  .pcase<case2>(expression2)
  ...
  .pelse_error(error_code);
```

where `integer_expression` is any Puffin expression that returns an integer for each iteration. The cases, `case1` and `case2`, must be Puffin constants. The expressions, `expression1` and `expression2`, can be any Puffin expression. There can be any number of cases, as long as the last case is `pelse_error`. A `pelse_error` clause takes a single integer argument.

For each iteration, the `integer_expression` is evaluated. Like with a true condition in `cond`, for the first clause that matches, the clause's expression is evaluated and returned as the result. If the `integer_expression`'s result does not match any clause's case, then `pelse_error` clause is immediately calls `exit` with `error_code` as the call's argument.

## 2.6 Reductions

Reductions are often used to find the sum or minimum value of an array or calculation. Currently, the only supported standalone reduction in Puffin is finding the minimum value of an expression: `puffin_min`, as that is all that is used in LULESH 2.0. The syntax for the minimum reduction, `puffin_min`, is:

```
puffin_min(aspect,
           expression,
           initial_value,
           result);
```

where `aspect` is a single aspect, `expression` is any Puffin expression that requires only `aspect`, `initial_value` is any scalar value, and result is a reference to a scalar value.

For a realistic example of standalone reductions, consider how LULESH 2.0 updates its hydro constraint each time step:

```
puffin_min(CurReg,
           cond(vdov != 0.0,
                dvovmax / (abs(vdov) +
                           1.0e-20))
               (dthydro),
           dthydro,
           dthydro);
```

As the syntax above shows, `puffin_min` is limited to ranging over a single aspect. While we do plan to extend Puffin to support reductions as dependent statements, Puffin already supports reductions as expressions.

For expression reductions, Puffin supports finding the sum, average, minimum value of values returned from a Puffin expression over an aspect's range.

```
reduce_over(aspect, expression)
```

where `reduce_over` is `sum_over`, `average_over`, or `min_over`; `aspect` is a single aspect, and `expression` is any Puffin expression that requires at least `aspect`. `expression` can also require any other aspects in the statement containing the reduction expression. The example from LULESH 2.0 in the next section shows a simple use of `sum_over`.

## 2.7 Assertions

Puffin provides assertions to report runtime errors easily. The syntax for a standalone Puffin assertion, `puffin_assert`, is similar to standalone reduction in Puffin:

```
puffin_assert(aspect,
              boolean_expression,
              error_code);
```

Like Puffin standalone reductions, this version of assertion is limited to ranging over a single aspect. Unlike Puffin standalone reductions, we have already extended Puffin to support assertions as dependent statements. The boolean expression parameter is any Puffin expression that returns a boolean value and requires only the aspect that is the first parameter. When executed, an assertion statement evaluates the given boolean expression over the range of the given aspect. If all boolean expressions in the aspect's range evaluate to true, the assertion finishes without any side effects. If any boolean expressions in the aspect's range evaluate to false, the assertion immediately calls the `exit` function with the `error_code` parameter as its only argument. Similar to `when` and `whenelse`, the dependent `assert` statement uses member function syntax.

The dependent statement version of assertions in Puffin has similar syntax and semantics:

```
puffin_foreach(aspect)
  ...
  .assert(boolean_expression,
          error_code)
  ...
  .execute();
```

For an example of a dependent assertion, consider how LULESH 2.0 checks to make sure the volume of all elements remains positive while also updating strains:

```
puffin_foreach(Elems)
  (vdov |= sum_over(Dim, ddd))
  (ddd[Dim] |= ddd - vdov / 3.0)
  .assert(vnew > 0.0, VolumeError)
  .execute();
```

where `ddd` is over the spatial dimensions and element aspects, and `vdov` and `vnew` are over the element aspect.

## 2.8 Affiliations

Puffin provides mechanisms for describing calculations over geometric/conceptual relationships between different types of arrays. In Puffin, these relationships are called **affiliations**. An affiliation is the relationship between each member of one aspect with one or more members of another aspect.

The most simple affiliation in LULESH 2.0 is between nodes and elements. In LULESH 2.0, elements are the unstructured volumes that partition the physical space in a simulation. Nodes are the corners of each element. Since all elements in LULESH 2.0 are hexahedrons, each element has eight corners. Thus, each element is affiliated with eight nodes. Conversely, each node is affiliated with a variable number of elements.

LULESH 2.0 represents these affiliations between elements and nodes as lists of indices. Puffin uses these affiliation lists in its own affiliation objects. Additionally like aspects, Puffin affiliations are used to group sets of calculations together. Conforming with the rest of Puffin, Puffin affiliations are divided into statements and expressions.

A Puffin affiliation used for a Puffin statement executes the statement once for each affiliated index. The syntax for a Puffin affiliation statement is the same as for a Puffin statement block. Consider the following example from LULESH 2.0's calculation of hourglass forces, with the Puffin affiliation from elements to nodes, `ElemToNode`:

```
puffin_foreach(Elems)
  (ElemToNode
    (fd[Dim] |= fd + hgfd))
  .execute();
```

where `fd` is an array over nodes and spatial dimensions aspects, and `hgfd` is over elements and spatial dimensions aspects as well as the affiliation aspect between elements and nodes. This code matches the original LULESH 2.0 code exactly, with its same drawback: It is not thread-safe.

A Puffin affiliation used for a Puffin expression groups the affiliated indices together into a single value. Like reductions, Puffin affiliation expressions find the sum of all values, the minimum value, the maximum value or the average value of all affiliated indices. Puffin affiliation expressions use member function syntax for the various types of affiliation expressions, respective of the list in the previous sentence: `sum`, `min`, `max`, and `average`. Consider again the example from LULESH 2.0's calculation of hourglass forces; this time with the Puffin affiliation from nodes to elements, `NodeToElem`:

```
fd[Nodes][Dim] <<= fd + NodeToElem.sum(hgfd);
```

This code matches the original LULESH 2.0 thread-safe code. The main drawback of this version is that it cannot be combined with the other hourglass force calculations because those are all over the element aspect, whereas this is over the node aspect.

One restriction common to both Puffin affiliation statements and expressions is that they both must be used with statements over the originating aspect.

## 3. IMPLEMENTATION

C++ template meta-programming is the basis of Puffin's implementation. Every valid Puffin expression and statement creates a templated object. The types of the operands, arguments, and components of each Puffin expression and statement are used as the template arguments to these objects.

## 3.1 Runtime Implementation

At runtime, the first thing a Puffin statement does is construct the proper objects (ASTs) as described above. For Puffin standalone statements, the syntax alone implies when object construction is complete. For Puffin statement blocks, the call to the `execute` member function at the end implies that object construction is complete. Once the objects for the expressions and statements are constructed, execution can begin. Execution begins by iterating over the aspects provided at the top level. Each aspect extends the index object (or creates an index object if one does not yet exist). The index object keeps track of all aspects currently being iterated over and the current position in that iteration space.

Iteration over a single aspect has the following form:

```
void iterate(Aspects aspects,
             Arguments arguments,
             Given given) {
  typedef typename Aspects::First::template
                   ExtendIndex<Given>::Index
           Index;
  int const size = aspects.first().size();
  for(int i = 0; i < size; i++) {
    Index index = aspects.first()
                        .extend_index(i,
                                      given);
    RecursiveType::iterate(aspects.rest(),
                           arguments,
                           index);
  };
};
```

where `aspects` are the aspects to range over, `given` is the current index object, `arguments` are the arguments that are specific to the statement currently being executed, `given` is the current index object, and `RecursiveType` is the type defined to handle recursively the iteration of the remaining aspects. Each iteration of the `for` loop, a new index is set up with the current iteration count for the current aspect.

Once all aspects have been set up in the current index object, the statement is executed. Each statement type is different, but the assignment statement is representative:

```
void iterate(Lhs & lhs,
             Rhs const & rhs,
             Index const & index) {
  lhs.ref(index) = rhs.eval(index);
};
```

where `lhs` is the Puffin array on left-hand-side of the assignment, `rhs` is the Puffin expression on the right-hand side of the assignment, and `index` is the current index object. The call to `lhs.ref(index)` returns a reference to the location in the `lhs` array represented by the current index object, and likewise, the call to `rhs.eval(index)` returns the result of the `rhs` expression evaluated at the current index.

Additionally, wherever aspects are directly referenced in Puffin code, such as dependent assignment, reductions, and affiliations, the index object is extended to the new aspects and iterated over. Nesting aspects and therefore loops like this allows Puffin code to be very flexible and adaptive to the calculations that are to be performed.

### 3.2 Current Optimizations

Currently, Puffin has a two major optimizations implemented. First and foremost is heavy use of templates. While C++ class inheritance could be used inside of Puffin to simplify the use of templates and the implementation overall, naive use of C++ inheritance requires use of virtual function lookup tables at runtime which imposes a high overhead cost. With Puffin's current implementation, we not only avoid the overhead of a virtual function lookup table but most Puffin functions can be inlined at compile-time, further reducing the runtime overhead of Puffin.

Second, we unroll many small aspect loops by hand. As with all loop unrolling, the goal is to reduce the loop overhead. Unlike general loop unrolling, with aspects of fixed size, we know the exact extend of the loop. Thus, we can completely unroll the loop. Additionally, by unrolling the loop we can move index information about specific iterations to compile-time. Thus, rather than having the iteration count of an aspect added to the index object as a runtime value, we can add the iteration count to index object as a compile-time-known value, i.e. a template argument. For certain Puffin objects, such as Puffin arrays and Puffin constants, knowing at compile-time which iteration count is being performed shifts runtime checks to compile-time and therefore reduces runtime overhead.

## 4. ADOPTION WORKFLOW

We designed Puffin from the outset to be used in existing C++98 projects, with an emphasis on hydrodynamic simulation projects. With this in mind, we designed Puffin to be easy to integrate and adopt into such projects. As such, Puffin is currently implemented in a collection of C++ header files and only uses standard C++98, with no compiler-specific directives or pragmas. Currently, Puffin has been tested to work with gcc, clang, and xlc. While we designed Puffin to work with the 1998 standard of C++, we know of no reason Puffin cannot work with the 2011 or 2014 standards of C++.

Adding Puffin to an existing C++98 is straightforward. First, one must make sure the Puffin header files are in a location that can be found by the compiler. Second, one must include the main Puffin header file wherever Puffin will be used:

```
  #include <Puffin.h>
```

Once we add parallel backends to Puffin, the process of incorporating these backends into existing projects will become more complicated. For example, our planned CUDA backend for Puffin will require NVIDIA's CUDA compiler to be added to existing projects' build systems. However, despite this complication for Puffin's planned parallel backends, we plan to keep the sequential version of Puffin simple and straightforward to add to any project.

Puffin adoption can be piecemeal. After Puffin has been added to a project, individual loops and calculations can be rewritten in Puffin. After each loops is rewritten into Puffin, correctness and performance tests can be run on the project as a whole to detect how each loop refactor affects overall results. Thus, conversion to Puffin can be done in developers' spare time, on new development efforts, and/or when debugging issues in existing code, while leaving well-established working code untouched.

Also, once several loops in a sequence are rewritten in Puffin, those Puffin blocks can be quickly and easily refactored. Because each Puffin statement block is extensible, it is easy to merge multiple blocks over the same or similar aspects into a single block. Moreover, it is just as easy to split a single block in to multiple blocks. Refactoring sequences of Puffin code into fewer loops code often reduces loop overhead. Unfortunately, these refactors can also increase register pressure and instruction usage per loop, which can cause register spilling and increase pressure on the instruction cache. Because of these mixed benefits, there are no hard and fast rules for optimizing Puffin code.

Finally, many existing software simulation projects have spent a large amount of resources testing and validating the results of their simulations. Switching platforms and/or rewriting the existing code can then become a more complicated and costly task because the new code must be validated again. A careful conversion to Puffin can simplify this validation effort, assuming that the existing project's results are trusted.

## 5. PRELIMINARY RESULTS

Puffin is a work in progress, and our goal for our initial prototype is to support everything needed to write a Puffin port of LULESH 2.0 [6]. Currently, Puffin is only a few features short of this goal. Because of Puffin's ability to be incrementally adopted, we were able to port most of LULESH 2.0 to Puffin. Out of LULESH 2.0's roughly 60 loops, including

| Version | Compile-time (sec) | Total LOC | Relative to | LOC Removed | LOC Added |
|---|---|---|---|---|---|
| Original | 13.39 | 3176 | — | — | — |
| Puffin Global Definitions | 13.95 | 3353 | Original | — | 177 |
| Simple Features | 21.44 | 3061 | Puffin Global Definitions | 703 | 355 |
| All Features | 39.69 | 2751 | Simple Features | 441 | 185 |
| Optimized Original | 10.61 | 3076 | Original | 267 | 167 |
| Optimized Global Definitions | 11.56 | 3253 | Optimized Original | — | 177 |
| Optimized Simple Features | 25.34 | 3033 | Optimized Global Definitions | 611 | 335 |
| | | | Simple Features | 75 | 47 |
| Optimized All Features | 38.37 | 2723 | Optimized Simple Features | 441 | 185 |
| | | | All Features | 75 | 47 |

**Table 1: Comparing the compile-times and lines of code (LOC) of different versions of LULESH 2.0. Original is the original version of LULESH 2.0 without MPI. Puffin Global Definitions is LULESH 2.0 with Puffin's global definitions of aspects, arrays, and global variables. Simple Features is the version of LULESH 2.0 using the "simple" features of Puffin, that is, all features except for Puffin match (Section 2.5.3) and Puffin affiliations (Section 2.8). All Features is the version of LULESH 2.0 using the all implemented features of Puffin, as described in Section 2. The Optimized versions of all of these have fused the 16 loops in `EvalEOSForElems` function into 2 loops. The Relative to column shows which other version the LOC Removed and LOC Added columns refer to for each row. Thus, the Optimized Simple Features version of LULESH 2.0 required 75 lines of code removed and 47 added, relative to the Simple Features version of LULESH 2.0.**

nested loops, only 16 cannot yet be ported to LULESH 2.0. Thus, all the results from the Puffin port of LULESH 2.0 below are from tests of all of LULESH 2.0 with nearly three fourths of its loops written in Puffin and the remaining loops are the original LULESH 2.0 code.

Modifying LULESH 2.0 to use Puffin is fairly straightforward. First, we define the aspects and the array types that we will use in LULESH 2.0. Next, we provide access to the quasi-global variables of LULESH 2.0's `Domain` class by building Puffin arrays for variables using the memory that LULESH 2.0 already allocates for them. All of these definitions take roughly 177 lines of code. In Table 1, the Puffin Global Definitions and Optimized Global Definitions represent adding these Puffin definitions to LULESH 2.0 without using Puffin anywhere. Having these definitions present slows down compilation by a small but noticeable amount, and does not impact runtime in a noticeable way.

The next task is rewriting parts of LULESH 2.0 using Puffin. While we replaced each loop one at a time, for the sake of space, we present just two versions here. The first version replaces all loops that use Puffin's "simple" features, that is, all loops which can be written in Puffin without using match, affiliations, or whole-aspect calculations. This version maintains LULESH 2.0's loop structure and is called Simple Features in Table 1. The second version of LULESH 2.0 with Puffin replaces all loops that use Puffin's implemented features, that is, all loops which can be written in Puffin without using whole-aspect calculations. (Whole-aspect calculations have not been implemented yet; see Section 7 for more details.) As Table 1 shows, in both versions, the Puffin code is about half the size of the code it replaces.

While Puffin can support three fourths of LULESH 2.0's loops, Puffin only replaces only about one third of LULESH 2.0's lines of code. The loops that contain whole-aspect calculations are very detailed calculations that require a lot more lines of code than do the loops that do not use whole-aspect calculations. Once we implement whole-aspect calculations in Puffin and port the rest of LULESH 2.0 to Puffin, we expect a more dramatic reduction in the total number of lines of code.

Then, we changed the loop structure of the `EvalEOSForElems` function in LULESH 2.0. The original version of this calculation is roughly 280 lines of code, uses three functions and 16 loops. The Puffin version of the calculation preserving loop structure takes about 140 lines of code, one function and 16 loops. By combining all 16 loops into two loops, the Puffin version takes about 120 lines of code, and its execution for all problem sizes speeds up by about 10%. (Avoiding data races prevents the last two loops to be merged.) For a fair comparison to LULESH 2.0, we optimized the original version of LULESH 2.0 with the same loop structure. In this optimized version of LULESH 2.0, the calculation took about 180 lines of code, and likewise its execution for all problem sizes speeds up by about 10%. (The test with a problem size of 30 speeds up by about 25%, bringing its execution to where we expect it to be.) In Table 1, the versions of LULESH 2.0 with optimized equations of state calculations are Optimized Original, Optimized Global Definitions, Optimized Simple Features, and Optimized All Features.

## 5.1 Maintaining Existing Results

Maintaining the same results during adoption of languages like Puffin is important to many existing software projects. Many software simulation projects have spent considerable time and effort validating the results of their simulation results. Re-validation increases the overall cost of adopting a new language or library that does not maintain the same results.

Throughout the process of porting LULESH 2.0 to Puffin, we strove to keep the same results as the original version of LULESH 2.0. For the most part, maintaining the results was straightforward: We simply maintained the same order of operations.

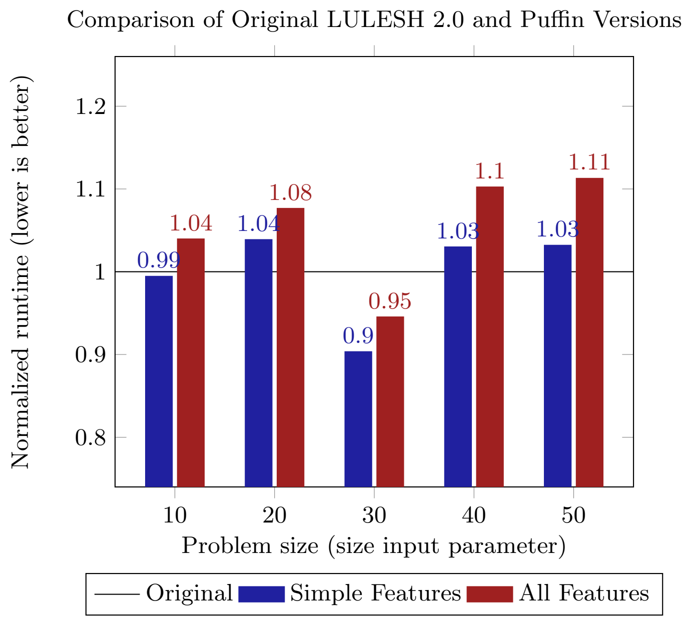

**Figure 1: Comparison of the Original version of LULESH 2.0 with the Simple Features and All Features versions. These numbers have been normalized based the Original version of LULESH 2.0. Thus, a time of 1.08 means that the All Features version ran 8% slower than the Original version of LULESH 2.0 for the test with problem size of 20. For the problem size 30 tests, the Original version runs slower than expected, which explains why the Puffin versions are 5-10% faster. See Figure 2 for more typical results.**

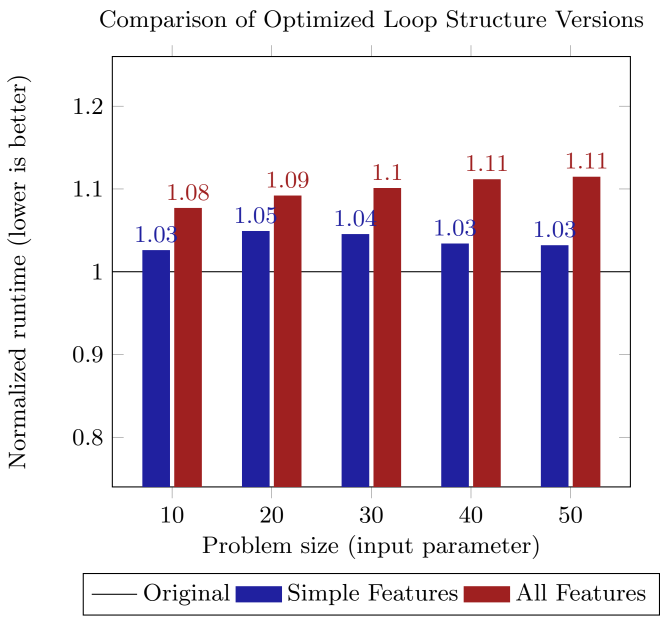

**Figure 2: Comparison of the Optimized Original version of LULESH 2.0 with the Optimized Simple Features and Optimized All Features versions. These numbers have been normalized based the Optimized Original version of LULESH 2.0. Thus, a time of 1.08 means that the All Features version ran 8% slower than the Original version of LULESH 2.0 for the test with problem size of 10.**

However, different compilers and different optimizations can produce different results because of extra-precision operations such as fused multiply-add (FMA). All versions of LULESH 2.0 return different results because of these extra-precision operations, when used with different compilers or different optimizations. Some compilers have flags that prevent the use of extra-precision operations, such as gcc's float-store flag. When compiled with these flags, all version of LULESH 2.0 presented in this paper produce the same results, down to the same rounding error. Without these flags, all version of LULESH 2.0 presented in this paper produce same energy at the origin, which is LULESH 2.0's standard correctness measurement, that ignores rounding error.

### 5.2 Performance

Figures 1 and 2 show the runtime overhead of Puffin using both the original and optimized loop structure. These tests were run on a BG/Q system with PPC A2 CPUs, using Clang version 3.7 (optimization level 3 and aggressive inlining). The "simple" version of Puffin has no more than 5% runtime overhead, and the "full" version of Puffin has no more than 11% runtime overhead.

The biggest cost to using Puffin is the compile-time overhead: As shown in Table 1, the "simple" version of Puffin takes nearly 2X time to compile, and the "full" version of Puffin takes almost 3X as long to compile. While this compile-time overhead is significant, most modern build systems support parallel compilation of multiple compilation units. This parallel compilation limits the impact of the compile-time overhead. Likewise, modern build systems only recompile compilation units whose source code has changed. Thus, during active development using Puffin, only the files that are modified must be recompiled.

## 6. RELATED WORK

There are many other domain-specific languages that have functionality and domains similar to Puffin, but none contain all of Puffin's features. Nebo [4] and Liszt [3] are probably the two closest projects. Nebo is a domain-specific language for solving partial differential equations on structured meshes. Nebo uses template meta-programming the same as Puffin. Nebo supports multiple backends, including a GPU backend. The main difference between them is that Puffin is targeting unstructured meshes, whereas Nebo is strictly limited to structured meshes.

Liszt [3] represents calculations by abstracting based on geometry and spatial reasoning, similar to Puffin affiliations. Liszt supports both CPU- and GPU-based parallelism, but does not support incremental adoption. Thus, entire applications must be written in Liszt to use Liszt.

RAJA [5] is an abstraction layer with similar goals to Puffin. Like Puffin, RAJA's main goal is modifying existing project so that the project is portable to multiple architectures. Unlike Puffin, RAJA focuses on changing as little about the existing project as possible. Thus, porting an existing project to RAJA will be faster and simpler than porting the same project to Puffin. However, code ported to Puffin will be easier to maintain and more flexible.

OptiMesh [7], developed with the Delite compiler [2], offers CPU- and GPU-based parallel backends within the same runtime environment, as we plan to add to Puffin. OptiMesh uses the same abstractions and much of the same syntax as Liszt. In general, OptiMesh performs better than Liszt because Delite supports more aggressive optimizations.

RAJA and OptiMesh support forms of incremental adoption, while Liszt does not. For OptiMesh, partial adoption require adding new compilers to a projects build system. In comparison, RAJA and Puffin works without adding a new compiler in existing C++ projects.

## 7. FUTURE WORK

Puffin requires more features, more backends, more developer tools, and more developer support before it is ready to be used in production projects.

First, we plan to implement the features needed to complete the Puffin port of LULESH 2.0. There are only two major features remaining to complete this goal. The largest, non-implemented feature is supporting whole-aspect functions. All functions Puffin currently supports all calculate a single scalar value from other scalar values. LULESH 2.0 requires vector- and matrix-based operations, such as cross product and calculating a determinant of a matrix. These calculation cannot be broken down into smaller scalar calculations based on the same indices.

The second feature to be implemented is better temporary array support. Currently, users must manually manage the temporary arrays used only within a single iteration of a Puffin statement block. While this approach works for now, these temporary arrays will not work with parallel backends.

Puffin's current main advantage is code that is shorter, more maintainable, and more flexible than current C++ code with low overhead. However, as we designed Puffin, our main goal was portable code. Our long-term plan, once Puffin's remaining features are done, is to implement a CUDA GPU backend to Puffin that will require minimal changes to the Puffin port of LULESH 2.0 to run efficiently on GPUs. Similar to Nebo's approach to GPU execution [4], each standalone statement and statement block will become a CUDA kernel. Beyond GPUs, we may implement backends targeting the Xeon Phi architecture family and parallel libraries such as OpenMP and MPI.

Currently, Puffin is a proof of concept: Abstractions like Puffin can simplify users' code without adding intolerable overhead. Once the work described above is complete, Puffin will be a prototype showing abstractions can create efficient portable code without unbearable costs.

However, this prototype will not yet be useful for production hydrodynamics codes. When this prototype is finished, it will still require testing, documentation, and debugging tools.

## 8. CONCLUSIONS

While Puffin is still just a prototype, we have begun to identify the costs and benefits of adopting Puffin in existing unstructured hydrodynamics simulation projects. While it is easy to add Puffin to an existing project's build system, there are costs to adopting Puffin. Puffin's highest initial cost is rewriting the existing code into Puffin. As our partial ports of LULESH 2.0 show, Puffin can be incrementally adopted. The recurring costs of compilation and runtime overhead cannot be avoided. Compilation is currently 2-3X slower, but with modern build systems this overhead can be reduced to acceptable levels. Runtime overhead is currently around or below 11%. While we will continue to try to reduce this overhead, the overhead is low enough that the benefits of a complete Puffin may be outweigh this overhead as it is now.

Currently, Puffin's benefits are limited, mainly because Puffin is not yet complete. Code written in Puffin is about half the size of loops it replaces. Additionally, we plan to add support for Puffin code on GPUs and other nontraditional architectures, which will provide an incrementally adoptable path to portable code for existing codebases.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Release Number: LLNL-CONF-676764

## 9. REFERENCES

[1] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[2] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100. IEEE, 2011.

[3] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[4] C. Earl. *Introspective Pushdown Analysis and Nebo.* School of Computing, University of Utah, 2014.

[5] R. D. Hornung and J. A. Keasler. The RAJA portability layer: Overview and status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

[6] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[7] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, et al. Composition and reuse with compiled domain-specific languages. In *Proceedings of ECOOP*, 2013.